MP24 @ II UWr 21 maja 2024 r.

Lista zadań nr 11

Zadanie 1. (1 pkt)

Naiwna implementacja funkcji `exists` z modułu `list` mogła by wyglądać następująco.

```
let exists f xs = List.fold_left (fun b -> b || f x) false xs
```

Niestety taka implementacja zawsze przegląda całą listę, nawet jeśli wynik jest znany już po sprawdzeniu pierwszego elementu. Używając funkcji `fold_left` zaimplementuj funkcję `exists` tak by nie miała tego mankamentu. Użyj wyjątków, by przerwać obliczenie gdy wynik jest już znany.

Zadanie 2. (1 pkt)

Używając wyjątków i funkcji `fold_left` zaimplementuj funkcję `find` (równoważną tej z modułu `List`). Podobnie jak w poprzednim zadaniu, funkcja powinna przerwać przeglądanie listy, gdy wynik już jest znany.

Wskazówka 1: Przeczytaj Wskazówkę 2. Nie przejmuj się, jeśli nie rozumiesz wszystkich używanych tam pojęć. Zaproponowane rozwiązanie możesz przyjąć na wiarę.

Wskazówka 2: choć ogólna idea rozwiązania tego zadania powinna być w miarę prosta, to główna trudność wynika z drobnych niezręczności w systemie typów OCamla. Będziemy potrzebować zdefiniować własny wyjątek, którego typ parametru zależy od konkretnego wywołania funkcji `find`. Można to osiągnąć tworząc nowy wyjątek dla każdego wywołania funkcji, używając *lokalnych wyjątków*.

```
let find p xs =
  let exception Found in
  ...
```

Problem się pojawia, gdy chcemy dodać parametr do tego wyjątku. Naiwne rozwiązanie nie działa:

```
let find p (xs : 'a list) =
  let exception Found of 'a in
  ...
```

```
Error: The type variable 'a is unbound in this type declaration.
```

Można ten problem rozwiązać używając *typów lokalnie abstrakcyjnych*, czyli wprowadzając konkretną nazwę dla parametru typowego, względem którego nasza funkcja jest polimorficzna.

```
let find (type t) p (xs : t list) =
  let exception Found of t in
  ...
```

Zadanie 3. (1 pkt)

Zmodyfikuj czysto-funkcyjny interpreter z wykładu tak, by błąd typu był wyjątkiem, który można obsłużyć wewnątrz interpretowanego języka. Zachowanie wyrażenia `(fun x -> x) + 42` powinno być takie samo jak wyrażenia `raise`.

Zadanie 4. (2 pkt)

Zmodyfikuj język z wykładu tak, by był więcej niż jeden rodzaj wyjątku. A dokładniej, niech wyrażenia `raise` oraz `try` zawierają dodatkowo etykietę (która nie musi nigdzie wcześniej być zadeklarowana) identyfikującą dany wyjątek. Etykiety, podobnie jak zmienne, mogą być identyfikatorami w składni konkretnej. Wyjątek rzucony przez `raise` powinien być obsłużony przez najbliższe wyrażenie `try` o takiej samej etykiecie. Na przykład, poniższe wyrażenie powinno się obliczyć do wartości 42.

```
try
  try raise A with
    B -> 13
with
  A -> 42
```

Zadanie 5. (2 pkt)

Zmodyfikuj język z wykładu tak, by wraz z rzucanym wyjątkiem były przekazywane dane. Teraz składnia abstrakcyjna wyrażenia `raise` zawiera wyrażenie, które powinno się obliczyć do wartości przekazywanej wraz z wyjątkiem, zaś wyrażenie `try` wiąże zmienną pod którą wartość z wyjątku zostanie podstawiona. Przykładowa składnia abstrakcyjna mogła by wyglądać następująco.

```
type expr =
  ...
  | Raise of expr
  | Try   of expr * ident * expr
```

Zaproponuj składnię konkretną, a następnie zmodyfikuj parser i interpreter. Jeśli rozwiązanie połączysz z rozwiązaniem poprzedniego zadania, będzie Ci łatwiej rozwiązać następne zadanie (będziesz miał tylko jeden język zamiast dwóch).

Zadanie 6. (2 pkt)

Napisz translację tłumaczącą program w języku z zadania 4 (gdzie jest wiele rodzajów wyjątków) do języka z zadania 5 (gdzie jest tylko jeden wyjątek, ale za to wraz z wyjątkiem można przekazać wartość).

Wskazówka: dla każdej etykiety wyjątku wygeneruj unikatową liczbę, którą przekażesz wraz z wyjątkiem. Handler wyjątku w wyrażeniu `try` może porównać otrzymaną etykietę (jako liczbę) z oczekiwaną. Jeśli są różne, wyjątek można rzucić ponownie. Jeśli zdecydowałeś się połączyć rozwiązania zadań 4 oraz 5, to mogą okazać się przydatne pary, które są zaimplementowane w języku: wraz z wyjątkiem można przekazać parę etykieta-wartość.

Zadanie 7. (2 pkt)

Dodaj wyjątki do języka IMP. Twój interpreter może być metacykliczny. Czy konstrukcje `try` i `raise` powinny być wyrażeniami, czy instrukcjami?

Zadanie 8. (1 pkt)

Powtórz poprzednie zadanie, tym razem razem rozszerzając czysto-funkcyjny interpreter.